Indexed Programming

Jeremy Gibbons

accu 2008

1. Collections

```
public interface List {
   void add (int index, Object element);
   boolean contains (Object o);
   Object get (int index);
   int size ();
}
```

Loss of precision:

```
List l = new ArrayList ();    //  ...which implements List
String s = "abc";
l.add (0, s);                 //  upcasting when inserting
s = (String) l.get (0);       //  downcasting when retrieving
```

2. Generics

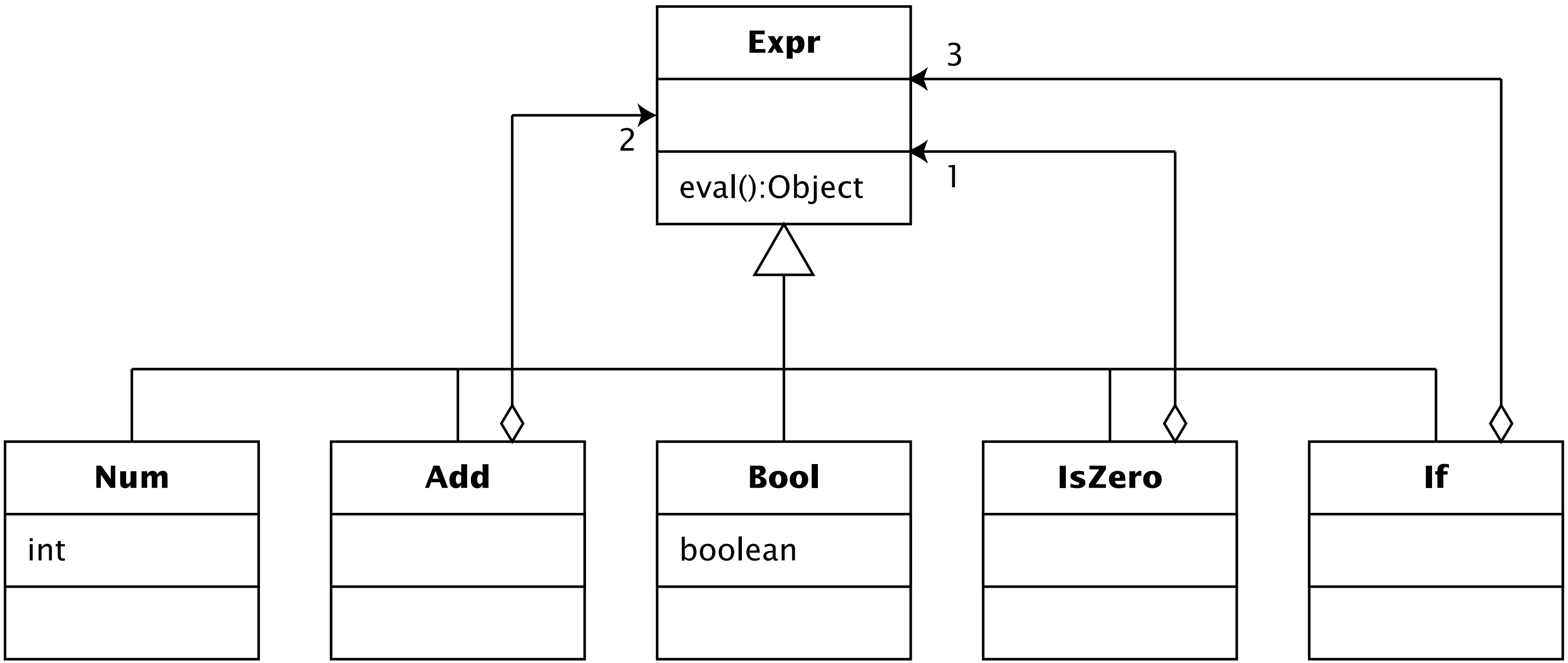
$$
\begin{array}{l}
\textbf{public interface } \mathit{List}\langle E\rangle\ \{\\
\quad \textbf{void } \mathit{add}\ (\textbf{int } \mathit{index}, E\ \mathit{element});\\
\quad \textbf{boolean } \mathit{contains}\ (\mathit{Object}\ o);\\
\quad E\ \mathit{get}\ (\textbf{int } \mathit{index});\\
\quad \textbf{int } \mathit{size}\ ();\\
\}
\end{array}
$$

More precise code:

$$
\begin{array}{ll}
\mathit{List}\langle \mathit{String}\rangle\ l = \mathit{new}\ \mathit{ArrayList}\langle \mathit{String}\rangle(); & \\
\mathit{String}\ s = \texttt{"abc"}; & \\
l.\mathit{add}\ (0, s); & //\quad \text{no upcasting or}\ldots\\
s = l.\mathit{get}\ (0); & //\quad \ldots\text{downcasting needed}
\end{array}
$$

This is called *parametric polymorphism*.

3. Algebraic datatypes



3.1. Expression datatype in Java

```
public abstract class Expr {
  public abstract Object eval ();
}

public class Num extends Expr {
  private Integer n;
  public Num (Integer n) { this.n = n; }
  public Object eval () { return n; }
}

public class Add extends Expr {
  private Expr x, y;
  public Add (Expr x, Expr y) {
    this.x = x; this.y = y;
  }
  public Object eval () {
    return (Integer) (x.eval ()) + (Integer) (y.eval ());
  }
}

public class Bool extends Expr {
  private Boolean b;
  public Bool (Boolean b) { this.b = b; }
  public Object eval () { return b; }
}
```

```
public class IsZero extends Expr {
  private Expr e;
  public IsZero (Expr e) { this.e = e; }
  public Object eval () {
    return new Boolean ((Integer) (e.eval ()) == 0);
  }
}

public class If extends Expr {
  private Expr b;
  private Expr t, e;
  public If (Expr b, Expr t, Expr e) {
    this.b = b; this.t = t; this.e = e;
  }
  public Object eval () {
    if (((Boolean) b.eval ()).booleanValue ())
      return t.eval ();
    else
      return e.eval ();
  }
}
```

3.2. Expression datatype in Haskell

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Expr} :: * \ \mathbf{where} \\
\quad \begin{array}{lll}
N & :: \mathit{Int} \to & \mathit{Expr} \\
B & :: \mathit{Bool} \to & \mathit{Expr} \\
\mathit{Add} & :: \mathit{Expr} \to \mathit{Expr} \to & \mathit{Expr} \\
\mathit{IsZ} & :: \mathit{Expr} \to & \mathit{Expr} \\
\mathit{If} & :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \to & \mathit{Expr}
\end{array} \\
\mathbf{data}\ \mathit{Result} = \mathit{NR}\ \mathit{Int} \mid \mathit{BR}\ \mathit{Bool} \\
\mathit{eval} :: \mathit{Expr} \to \mathit{Result} \\
\begin{array}{ll}
\mathit{eval}\ (N\ n) & = \mathit{NR}\ n \\
\mathit{eval}\ (B\ b) & = \mathit{BR}\ b \\
\mathit{eval}\ (\mathit{Add}\ x\ y) & = \mathbf{case}\ (\mathit{eval}\ x, \mathit{eval}\ y)\ \mathbf{of}\ (\mathit{NR}\ m, \mathit{NR}\ n) \to \mathit{NR}\ (m + n) \\
\mathit{eval}\ (\mathit{IsZ}\ x) & = \mathbf{case}\ (\mathit{eval}\ x)\ \mathbf{of}\ \mathit{NR}\ n \to \mathit{NB}\ (0 \equiv n) \\
\mathit{eval}\ (\mathit{If}\ x\ y\ z) & = \mathbf{case}\ (\mathit{eval}\ x)\ \mathbf{of}\ \mathit{NB}\ b \to \mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}\ y\ \mathbf{else}\ \mathit{eval}\ z
\end{array}
\end{array}
$$

4. Indexing

Loss of precision again: expressions of different ‘types’.

Note the explicit tagging and untagging in Haskell, and the casts in Java. (And the lack of error-checking for ill-formed expressions!)

Can we capture the precise constraints in code, and exploit them?

4.1. Indexed datatypes

Parametrise the datatype (where parameter expresses represented type):

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Expr} :: * \to * \ \mathbf{where} \\
\quad \begin{array}{lll}
N & :: \mathit{Int} \to & \mathit{Expr}\ \mathit{Int} \\
B & :: \mathit{Bool} \to & \mathit{Expr}\ \mathit{Bool} \\
\mathit{Add} & :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to & \mathit{Expr}\ \mathit{Int} \\
\mathit{IsZ} & :: \mathit{Expr}\ \mathit{Int} \to & \mathit{Expr}\ \mathit{Bool} \\
\mathit{If} & :: \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ a \to \mathit{Expr}\ a \to & \mathit{Expr}\ a
\end{array}
\end{array}
$$

Note that the parameter denotes a *phantom type*:
a value of type $\mathit{Expr}\ a$ need not contain elements of type a.

4.2. Indexed programming

Specialised return types of constructors induce type constraints, which are exploited in type-checking definitions.

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr}\ a \rightarrow a\\
&\mathit{eval}\ (N\ n) &&= n\\
&\mathit{eval}\ (B\ b) &&= b\\
&\mathit{eval}\ (\mathit{Add}\ x\ y) &&= \mathit{eval}\ x + \mathit{eval}\ y\\
&\mathit{eval}\ (\mathit{IsZ}\ x) &&= 0 \equiv \mathit{eval}\ x\\
&\mathit{eval}\ (\mathit{If}\ x\ y\ z) &&= \mathbf{if}\ \mathit{eval}\ x\ \mathbf{then}\ \mathit{eval}\ y\ \mathbf{else}\ \mathit{eval}\ z
\end{aligned}
$$

Note that all the tagging and untagging has gone, and with it the possibility of run-time errors.

By explicitly stating a property formerly implicit in the code, we have gained both in safety and in efficiency.

4.3. Indexed programming in Java

It can be done with Java (or C#) generics, but it's not so pretty:

```
public class If<T> extends Expr<T> {
  private Expr<Boolean> b;
  private Expr<T> t, e;
  public If (Expr<Boolean> b, Expr<T> t, Expr<T> e) {
    this.b = b; this.t = t; this.e = e;
  }
  public T eval () {
    if (b.eval ().booleanValue ()) return t.eval (); else return e.eval ();
  }
}
```

(Actually, not quite all the checking can be done at compile-time; sometimes some casts are still necessary.)

5. Other applications

Indexing by:

size: eg bounded vectors

shape: eg red-black trees

state: eg locking of resources

unit: eg physical dimensions

type: eg datatype-generic programming

proof: eg web applets

6. Application: indexing by size

Empty datatypes as indices (so $S\ (S\ Z)$ is a type).

$$
\begin{array}{l}
\textbf{data}\ Z \\
\textbf{data}\ S\ n
\end{array}
$$

Size-indexed type of vectors:

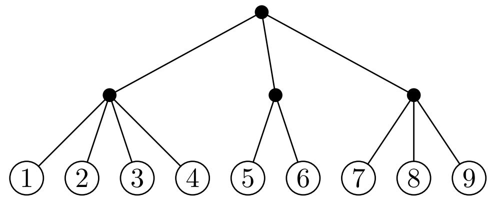
$$
\begin{array}{lll}
\textbf{data}\ \mathit{Vector} :: * \to * \to *\ \textbf{where} & & \\
\quad \mathit{VNil} & :: & \mathit{Vector}\ a\ Z \\
\quad \mathit{VCons} & :: a \to \mathit{Vector}\ a\ n \to & \mathit{Vector}\ a\ (S\ n)
\end{array}
$$

Size constraint on *vzip* is captured in the type:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{vzip} :: \mathit{Vector}\ a\ n \to \mathit{Vector}\ b\ n \to \mathit{Vector}\ (a, b)\ n} \\
\mathit{vzip}\ \mathit{VNil}\ \mathit{VNil} & = \mathit{VNil} \\
\mathit{vzip}\ (\mathit{VCons}\ a\ x)\ (\mathit{VCons}\ b\ y) & = \mathit{VCons}\ (a, b)\ (\mathit{vzip}\ x\ y)
\end{array}
$$

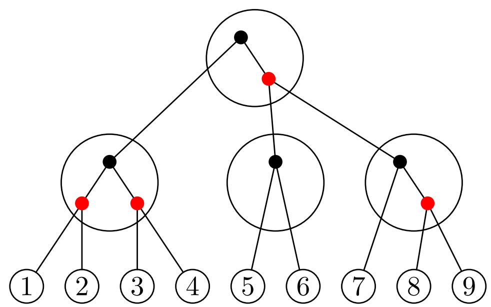
7. Application: indexing by shape

2-3-4 trees are perfectly-balanced search trees.



Representable as *red-black trees* — binary search trees in which:

- every node is coloured either red or black
- every red node has a black parent
- every path from the root to a leaf contains the same number of black nodes (enforcing approximate balance)



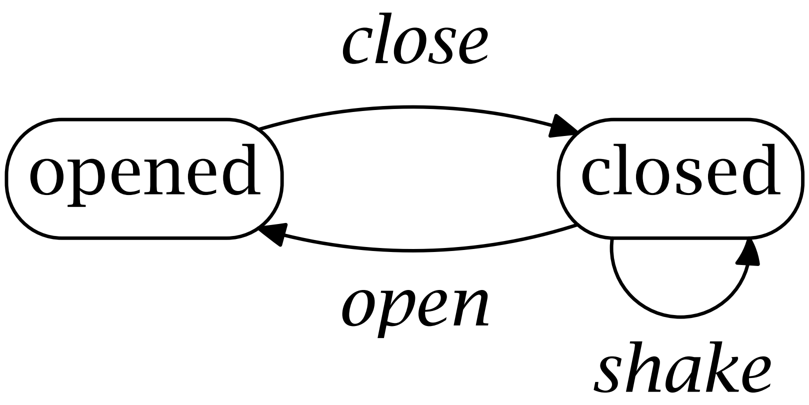
In $\mathit{RBTree}\ a\ c\ n$,

- a is the element type;
- c is the root colour;
- n is the black height.

$$
\begin{array}{l}
\textbf{data}\ R \\
\textbf{data}\ B \\
\textbf{data}\ \mathit{RBTree} :: * \to * \to * \to * \ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Empty} & :: & \mathit{RBTree}\ a\ B\ Z \\
\mathit{Red} & :: \mathit{RBTree}\ a\ B\ n \to a \to \mathit{RBTree}\ a\ B\ n & \to \mathit{RBTree}\ a\ R\ n \\
\mathit{Black} & :: \mathit{RBTree}\ a\ c\ n \to a \to \mathit{RBTree}\ a\ c'\ n & \to \mathit{RBTree}\ a\ B\ (S\ n)
\end{array}
\end{array}
$$

8. Application: indexing by state

The 'ketchup problem':



$$\begin{array}{l}
\mathbf{data}\ O \\
\mathbf{data}\ C \\
\mathbf{data}\ \mathit{Edge} :: * \to * \to *\ \mathbf{where} \\
\quad \mathit{Open} \ \ :: \mathit{Edge}\ O\ C \\
\quad \mathit{Close} \ \ :: \mathit{Edge}\ C\ O \\
\quad \mathit{Shake} :: \mathit{Edge}\ C\ C \\
\mathbf{data}\ \mathit{Path} :: * \to * \to *\ \mathbf{where} \\
\quad \mathit{Empty} :: \mathit{Path}\ s\ s \\
\quad \mathit{PCons} :: \mathit{Edge}\ x\ y \to \mathit{Path}\ y\ z \to \mathit{Path}\ x\ z \\
\mathit{scenario} :: \mathit{Path}\ O\ O \\
\mathit{scenario} = \mathit{PCons}\ \mathit{Open}\ (\mathit{PCons}\ \mathit{Shake}\ (\mathit{PCons}\ \mathit{Close}\ \mathit{Empty}))
\end{array}$$

9. Application: indexing by unit

Suppose dimensions of non-negative powers of metres and seconds:

$$\textbf{data}\ \mathit{Dim} :: * \to * \to *\ \textbf{where}$$
$$\quad D :: \mathit{Float} \to \mathit{Dim}\ m\ s$$

$$\mathit{distance} :: \mathit{Dim}\ (S\ Z)\ Z$$
$$\mathit{distance} = D\ 3.0$$

$$\mathit{time} :: \mathit{Dim}\ Z\ (S\ Z)$$
$$\mathit{time} = D\ 2.0$$

A dimensioned value is a *Float* with two type-level tags.

$$\mathit{dadd} :: \mathit{Dim}\ m\ s \to \mathit{Dim}\ m\ s \to \mathit{Dim}\ m\ s$$
$$\mathit{dadd}\ (D\ x)\ (D\ y) = D\ (x + y)$$

Now *dadd time time* is well-typed, but *dadd distance time* is ill-typed.

(More interesting to allow negative powers too, but for brevity. . .)

9.1. Type-level functions

Proofs of properties about indices:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Add} :: * \rightarrow * \rightarrow * \rightarrow *\ \mathbf{where} \\
\quad \mathit{AddZ} :: \phantom{\mathit{Add}\ m\ n\ p \rightarrow{}} \mathit{Add}\ Z\ n\ n \\
\quad \mathit{AddS} :: \mathit{Add}\ m\ n\ p \rightarrow \mathit{Add}\ (S\ m)\ n\ (S\ p)
\end{array}
$$

Used to constrain the type of dimensioned multiplication:

$$
\begin{array}{l}
\mathit{dmult} :: (\mathit{Add}\ m_1\ m_2\ m, \mathit{Add}\ s_1\ s_2\ s) \rightarrow \\
\qquad\qquad \mathit{Dim}\ m_1\ s_1 \rightarrow \mathit{Dim}\ m_2\ s_2 \rightarrow \mathit{Dim}\ m\ s \\
\mathit{dmult}\ (\_, \_)\ (D\ x)\ (D\ y) = D\ (x \times y)
\end{array}
$$

Thus, type-index of product is computed from indices of arguments.

9.2. Inferring proofs of properties

Capture the proof as a type class (multi-parameter, with functional dependency; essentially a function on types).

$$\begin{array}{l}
\textbf{class}\ \mathit{Add}\ m\ n\ p \mid m\ n \rightarrow p \\
\textbf{instance}\ \phantom{\mathit{Add}\ m\ n\ p \Rightarrow}\ \mathit{Add}\ Z\ n\ n \\
\textbf{instance}\ \mathit{Add}\ m\ n\ p \Rightarrow \mathit{Add}\ (S\ m)\ n\ (S\ p)
\end{array}$$

Now the proof can be (type-)inferred rather than passed explicitly.

$$\begin{array}{l}
\mathit{dmult} :: (\mathit{Add}\ m_1\ m_2\ m, \mathit{Add}\ s_1\ s_2\ s) \Rightarrow \\
\qquad\qquad \mathit{Dim}\ m_1\ s_1 \rightarrow \mathit{Dim}\ m_2\ s_2 \rightarrow \mathit{Dim}\ m\ s \\
\mathit{dmult}\ (D\ x)\ (D\ y) = D\ (x \times y)
\end{array}$$

Note that the type class has no methods, so corresponds to an empty dictionary; it can be optimized away.

10. Application: indexing by type

Generic programming is about writing programs parametrized by datatypes; for example, a generic data marshaller.

One implementation of generic programming manifests the parameters as some family of *type representations*.

For example, C's *sprintf* is generic over a family of *format specifiers*.

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Format} :: * \to * \ \mathbf{where} \\
\quad I :: \qquad\qquad\quad \mathit{Format}\ a \to \mathit{Format}\ (\mathit{Int} \to a) \\
\quad B :: \qquad\qquad\quad \mathit{Format}\ a \to \mathit{Format}\ (\mathit{Bool} \to a) \\
\quad S :: \mathit{String} \to \mathit{Format}\ a \to \mathit{Format}\ a \\
\quad F :: \qquad\qquad\qquad\qquad\qquad\quad \mathit{Format}\ \mathit{String}
\end{array}
$$

A term of type $\mathit{Format}\ a$ is a representation of the type a,
for various types a (such as $\mathit{Int} \to \mathit{Bool} \to \mathit{String}$) appropriate for *sprintf*.

10.1. Type-indexed dispatching

The function *sprintf* *interprets* the representation, generating a function of the appropriate type:

$$
\begin{array}{l}
\mathit{sprintf} :: \mathit{Format}\ a \to a \\
\mathit{sprintf}\ \mathit{fmt} = \mathit{aux}\ \mathit{id}\ \mathit{fmt}\ \textbf{where} \\
\quad \mathit{aux} :: (\mathit{String} \to \mathit{String}) \to \mathit{Format}\ a \to a \\
\quad \mathit{aux}\ f\ (I\ \mathit{fmt}) \quad\;\; = \lambda n \to \mathit{aux}\ (f \circ (\mathit{show}\ n +\!\!+))\ \mathit{fmt} \\
\quad \mathit{aux}\ f\ (B\ \mathit{fmt}) \quad\; = \lambda b \to \mathit{aux}\ (f \circ (\mathit{show}\ b +\!\!+))\ \mathit{fmt} \\
\quad \mathit{aux}\ f\ (S\ s\ \mathit{fmt}) = \mathit{aux}\ (f \circ (s +\!\!+))\ \mathit{fmt} \\
\quad \mathit{aux}\ f\ (F) \qquad\quad\; = f\ \texttt{""}
\end{array}
$$

For example, $\mathit{sprintf}\ f\ 13\ \mathit{True} = \texttt{"Int is 13, bool is True."}$, where

$$
\begin{array}{l}
f :: \mathit{Format}\ (\mathit{Int} \to \mathit{Bool} \to \mathit{String}) \\
f = S\ \texttt{"Int is "}\ (I\ (S\ \texttt{", bool is "}\ (B\ (S\ \texttt{"."}\ F))))
\end{array}
$$

11. Application: indexing by proof

The game of *Mini-Nim*:

- a pile of matchsticks
- players take turns to remove one match or two
- player who removes the last match wins

Index positions by size and proof of destiny.

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Win} \\
\mathbf{data}\ \mathit{Lose} \\[1ex]
\mathbf{data}\ \mathit{Position}\ n\ r\ \mathbf{where} \\
\quad \begin{array}{lll}
\mathit{Empty} & :: & \mathit{Position}\ Z\ \mathit{Lose} \\
\mathit{Take}1 & :: & \mathit{Position}\ n\ \mathit{Lose} \rightarrow \mathit{Position}\ (S\ n)\ \mathit{Win} \\
\mathit{Take}2 & :: & \mathit{Position}\ n\ \mathit{Lose} \rightarrow \mathit{Position}\ (S\ (S\ n))\ \mathit{Win} \\
\mathit{Fail} & :: & \mathit{Position}\ n\ \mathit{Win} \rightarrow \mathit{Position}\ (S\ n)\ \mathit{Win} \rightarrow \mathit{Position}\ (S\ (S\ n))\ \mathit{Lose}
\end{array}
\end{array}
$$

12. Adding weight

We have shown some examples in Haskell with small extensions.

This is a very lightweight approach to dependently-typed programming.

Lightweight approaches have low entry cost, but relatively high continued cost: encoding via type classes etc is a bit painful.

Tim Sheard's *Ωmega* is a cut-down version of Haskell with explicit support for GADTs:

- kind declarations
- type-level functions
- statically-generated witnesses

Xi and Pfenning's *Dependent ML* provides natural-number indices, and incorporates decision procedures for discharging proof obligations.

These are more heavyweight approaches (such as McBride et al's *Epigram*).

13. Conclusions

- generics have become mainstream
- ...but so much more than parametric polymorphism!
- *indexed programming*: a form of lightweight *dependent types*
- *Generic and Indexed Programming* project at Oxford
- perhaps algebraic datatypes will jump the gap next? (cf Scala)

14. Shameless plug...

Software Engineering at Oxford | Welcome to the Software Engineering Programme

http://www.softeng.ox.ac.uk/

UNIVERSITY OF OXFORD
SOFTWARE ENGINEERING PROGRAMME
PART-TIME POSTGRADUATE STUDY

Contact us | Search | Site map | Login

Programme: About, History, People, Events, Contact
Courses: About, Subjects, Calendar, Booking
Study: About, Awards, Fees, Applications
Research: About, Projects, People, Seminars, Collaboration
Resources: 2008 Brochure, Documents, Forms

Welcome to the Software Engineering Programme

Welcome to the Software Engineering Programme, a programme of advanced education and applied research at the University of Oxford, offering access to the combined expertise and resources of the Oxford University Computing Laboratory and the Department for Continuing Education. The Programme has been supported by the Engineering and Physical Sciences Research Council (EPSRC) since 1992.

The Programme offers opportunities for part-time study, with courses in key areas such as object technology, software architecture, computer security, precise modelling, and development processes. Each course is delivered by an expert in the subject, and includes an intense teaching week of classes, practicals, and group work; class sizes are kept small to facilitate interaction and to promote learning.

These courses may be used as credit towards postgraduate qualifications—at Certificate, Diploma and Masters' (MSc) level—from the University of Oxford. These qualifications are open to those with an appropriate combination of academic and industrial experience; a first degree in computer science or software engineering is welcome, but not necessary.

The Programme is also a centre for research activity. The teaching staff are involved in a number of national and international projects in the areas of: digital mammography; cancer clinical trials informatics; distributed computing for climate prediction; languages and tools for object model transformation; virtual research environments; model-driven software engineering.

Read more about the Programme, short courses, part-time study, or research.

Done